Discussion Session 6

Sikun LIN
sikun@ucsb.edu

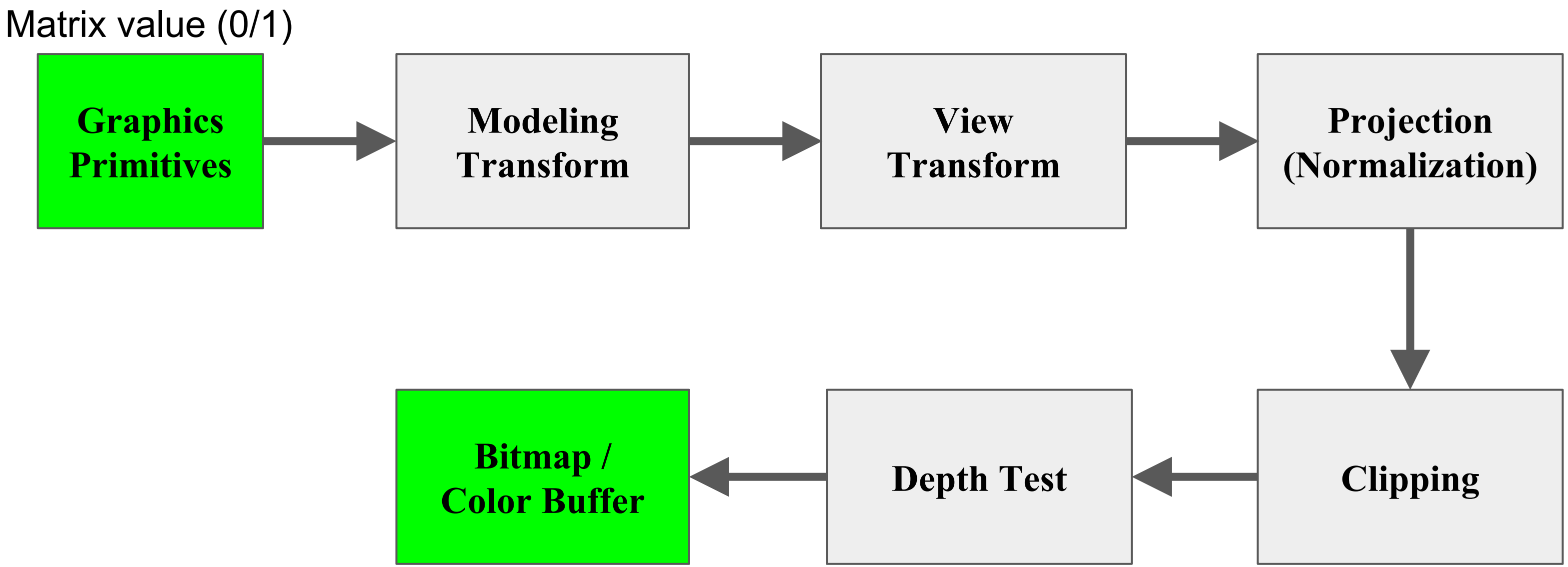
Today’s topic: things you need for hw3 (I)

- Matrices
- Polygon clipping
- HLHSR

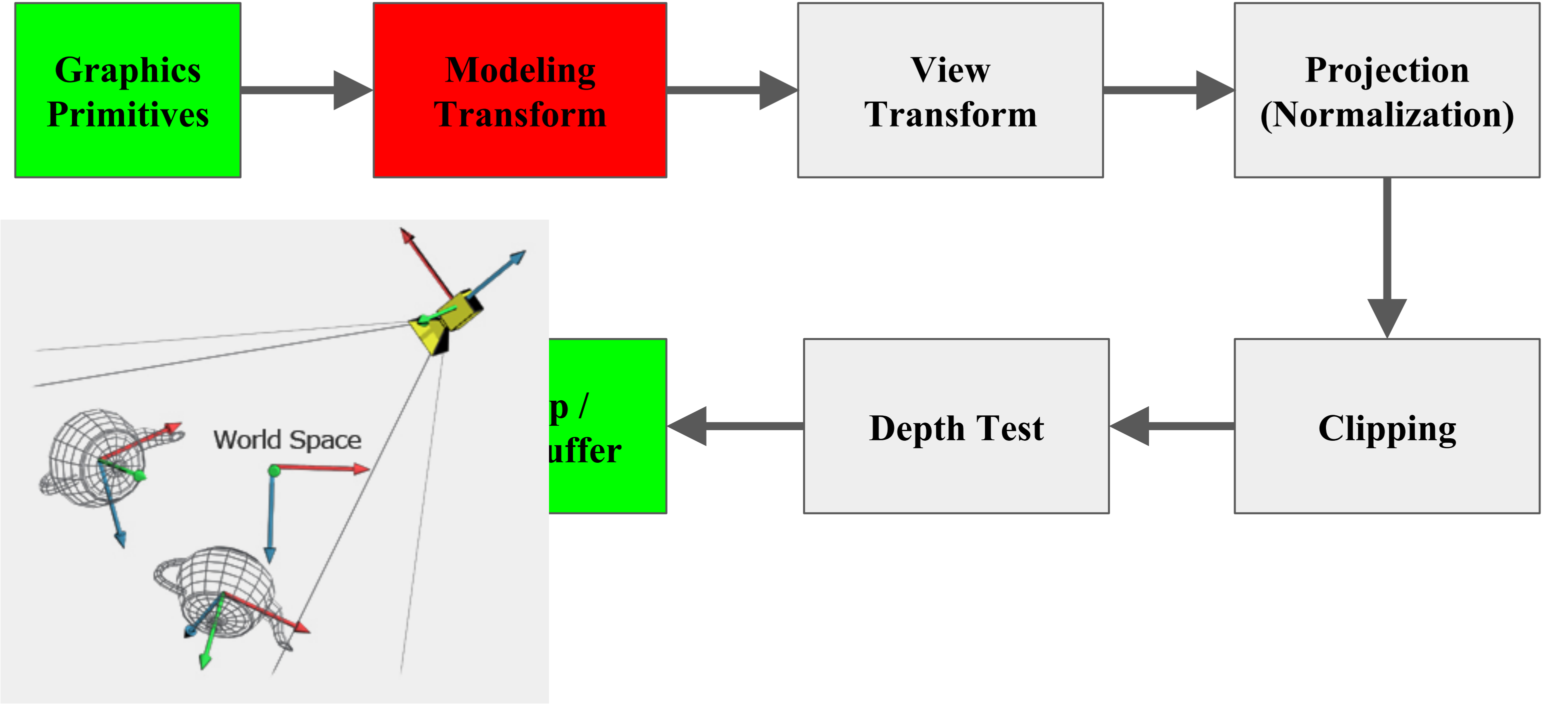
HW3 Requirements

- Implement your own graphics library:
  - Object drawing in black & white
  - Animation
  - Extra credit: viewpoint change
  - No color/Lighting/shading/texture/shadow
  - **CAN’T use OpenGL or any other graphics libraries**
  - Save each frame as an image file (ex. JPG, PPM) (can use library for writing/saving image)
- What you should turn in
  - All your code
  - Makefile, which can generate a series of image files for all frames
  - A video sequence or gif showing the final display result
    - you can use any software or free-use website to concatenate those frames

Rendering Pipeline



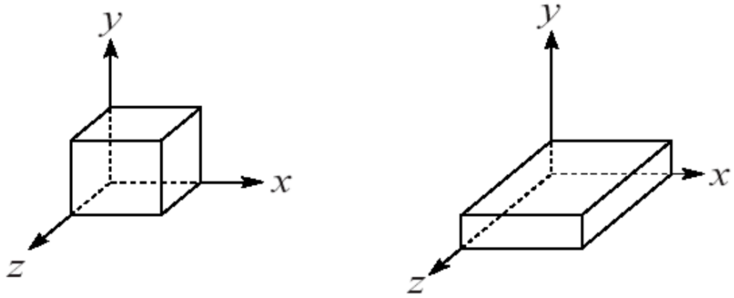
Rendering Pipeline



You need to implement

- Scaling
- Translation
- Rotation

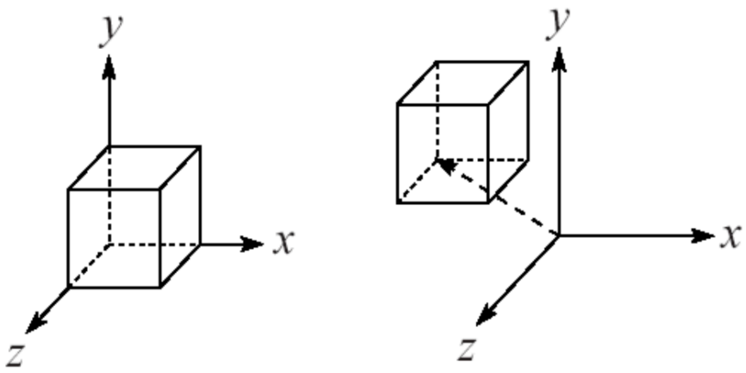
$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$



You need to implement

- Scaling
- Translation
- Rotation

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$



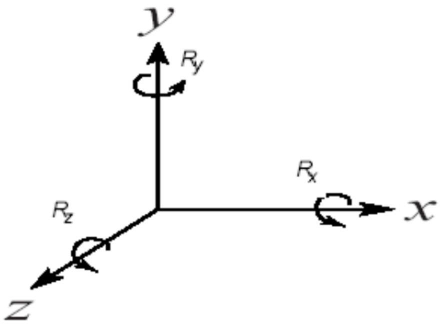
You need to implement

- Scaling
- Translation
- Rotation

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

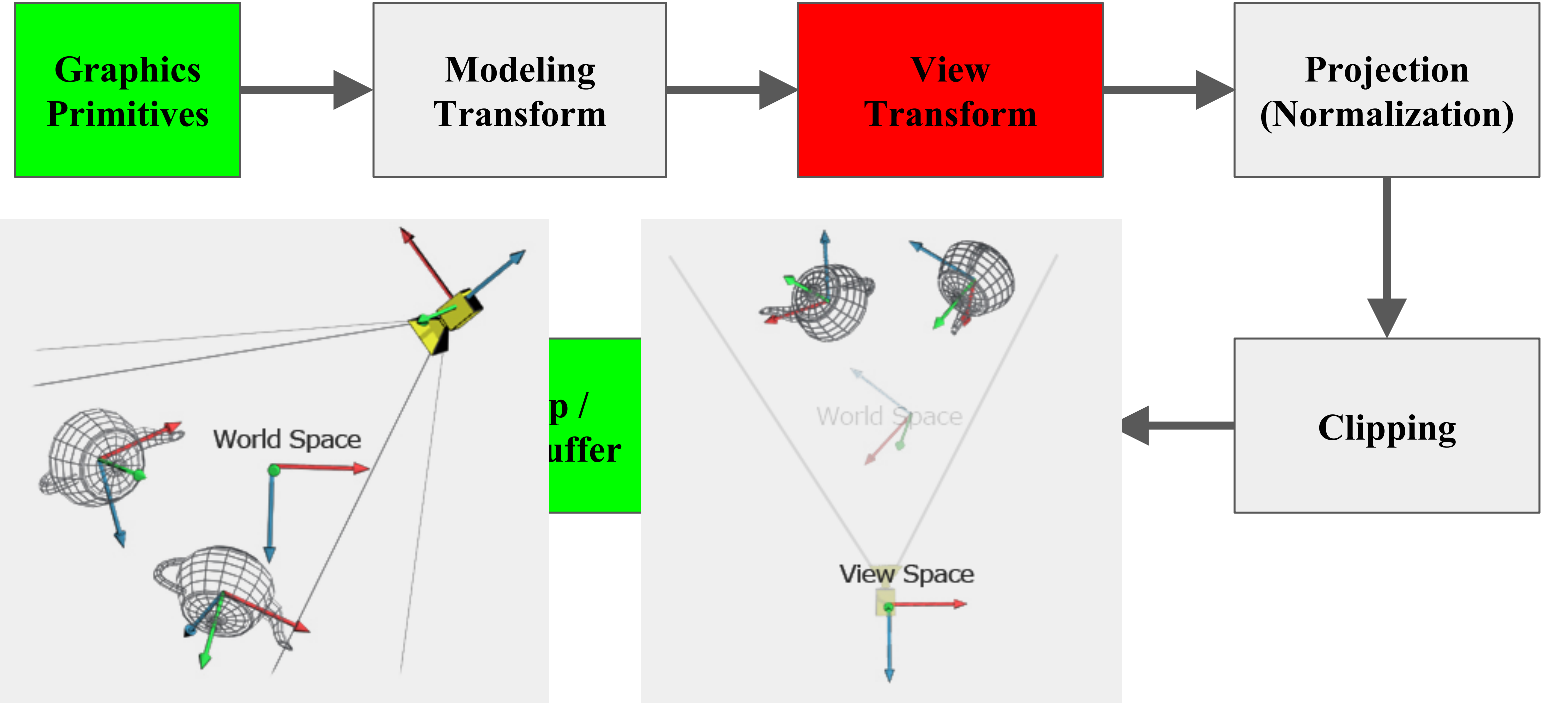
$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Use right hand rule

$$R = R_z(\theta_z)R_y(\theta_y)R_x(\theta_x)$$

Rendering Pipeline



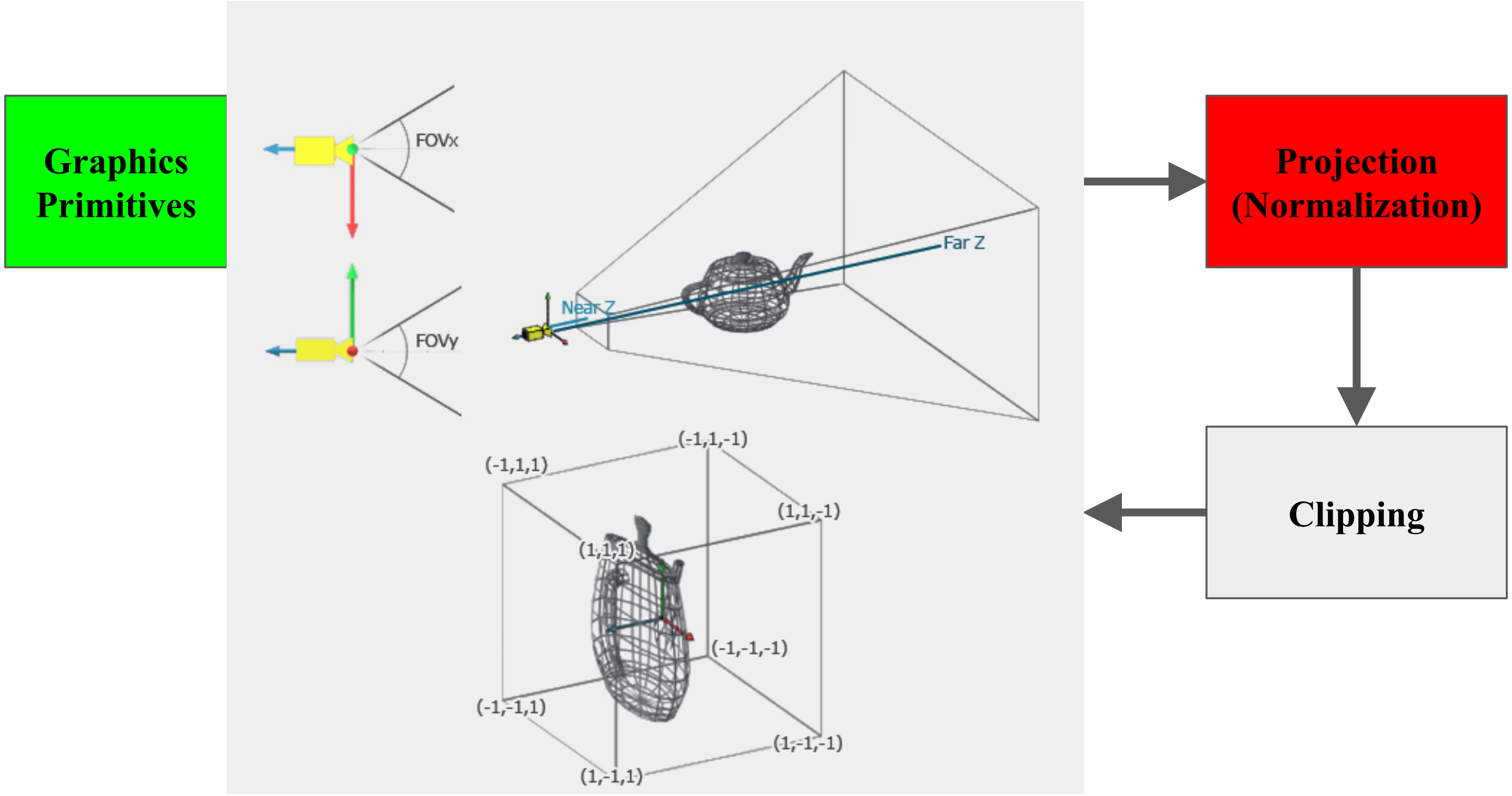
You need …

- Viewing matrix
- Knowns: eye position ***e***, center ***c***, up vector ***u***

$$\boldsymbol{f} = \boldsymbol{c} - \boldsymbol{e}$$
$$\boldsymbol{f}' = \frac{\boldsymbol{f}}{|\boldsymbol{f}|}$$
$$\boldsymbol{u}' = \frac{\boldsymbol{u}}{|\boldsymbol{u}|}$$
$$\boldsymbol{s} = \boldsymbol{f}' \times \boldsymbol{u}'$$
$$\boldsymbol{u}'' = \frac{\boldsymbol{s}}{|\boldsymbol{s}|} \times \boldsymbol{f}'$$

$$M = \begin{pmatrix} s_x & s_y & s_z & 0 \\ u''_x & u''_y & u''_z & 0 \\ -f'_x & -f'_y & -f'_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rendering Pipeline



You need …

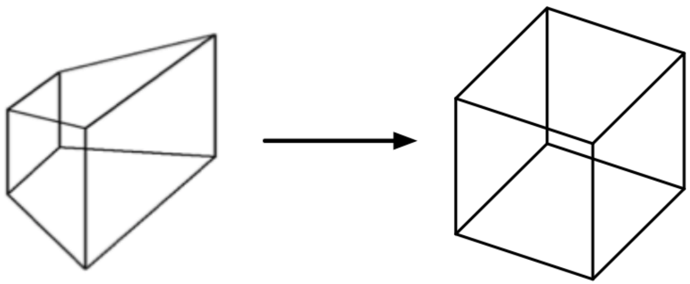
- Projection matrices (already transformed into canonical view volume)
- Perspective & orthographic (6 parameters for both: left right top bottom near far)

$$\begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \qquad \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{(f-n)} & -\frac{f+n}{(f-n)} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Canonical view volume

It can be computationally expensive to check if a point is inside a frustum

- Instead transform the frustum into a normalised canonical view volume
- Uses the same ideas a perspective projection
- Makes clipping and hidden surface calculation much easier

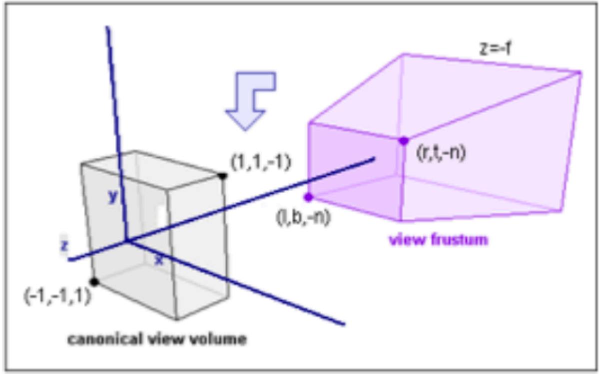


Transforming the view frustum

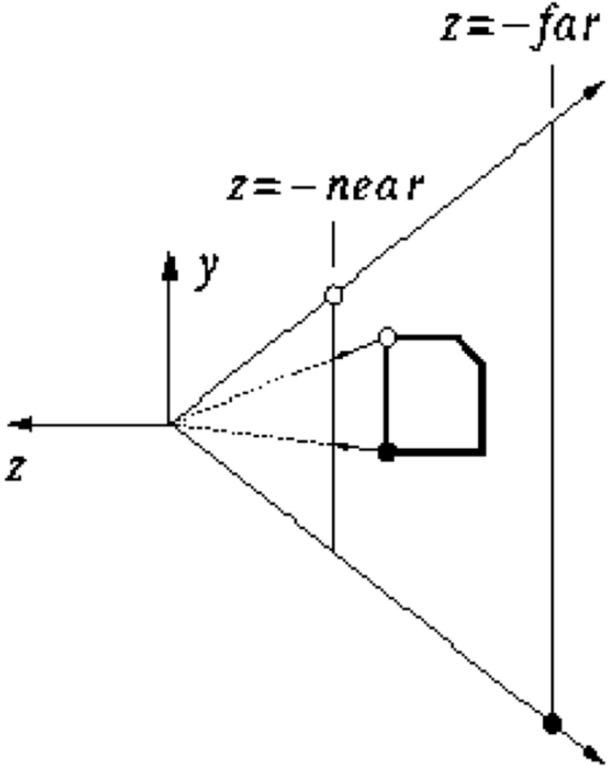
The frustum is defined by a set of parameters, l, r, b, t, n, f:

l	Left x coordinate of near plane
r	Right x coordinate of near plane
b	Bottom y coordinate of near plane
t	Top y coordinate of near plane
n	Minus z coordinate of near plane
f	Minus z coordinate of far plane

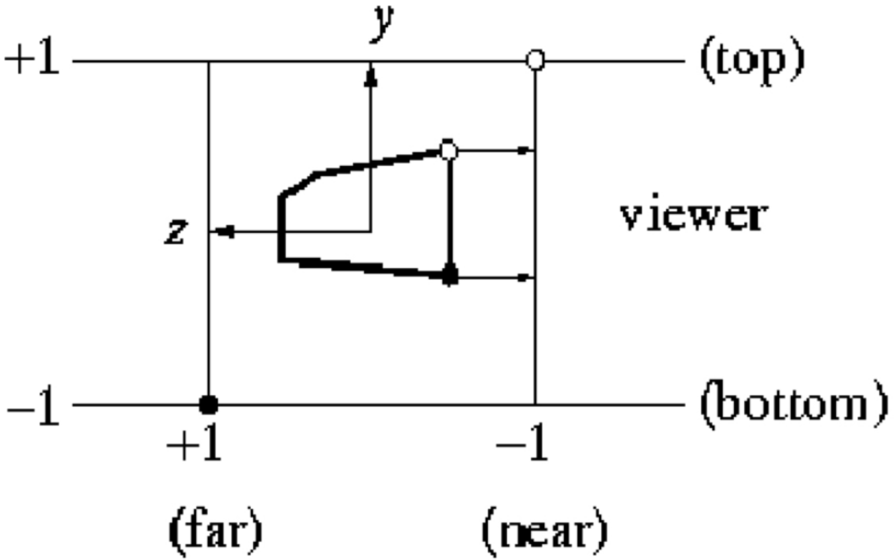
With $0 < n < f$.



$$-1 \leq x \leq 1, -1 \leq y \leq 1, -1 \leq z \leq 1$$



Viewing frustum

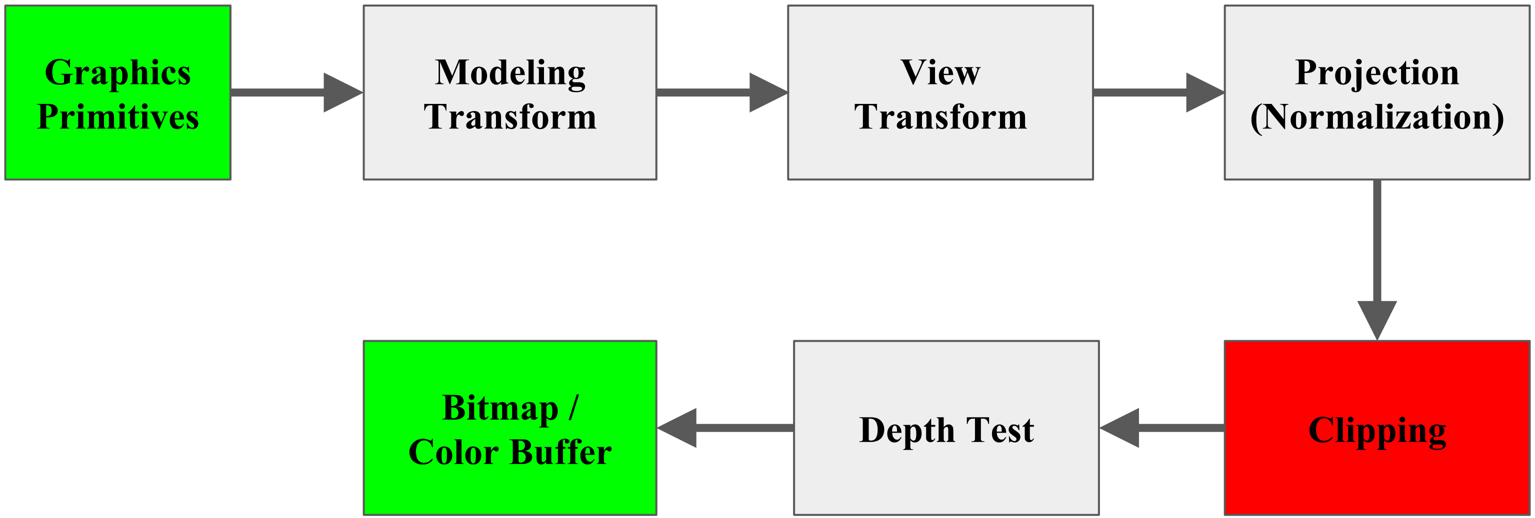


Canonical view volume

matrix.c

- Has all the matrix operations you need:
  - Inverse
  - Transpose
  - Addition/subtraction/multiplication
  - Inner/cross product
  - Determinant
  - …

Rendering Pipeline

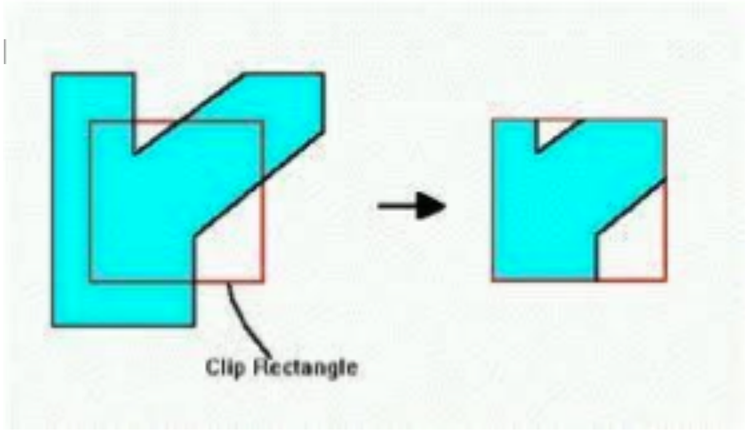


Clipping

● Perform clipping in the canonical view volume

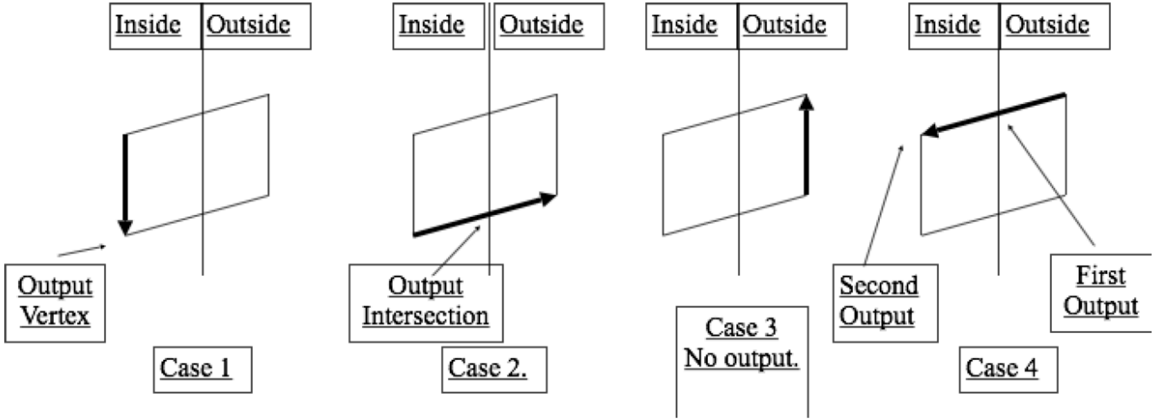
● Polygons may intersect the canonical view volume, then we need to perform clipping:

● Sutherland-Hodgma



Sutherland-Hodgman algorithm

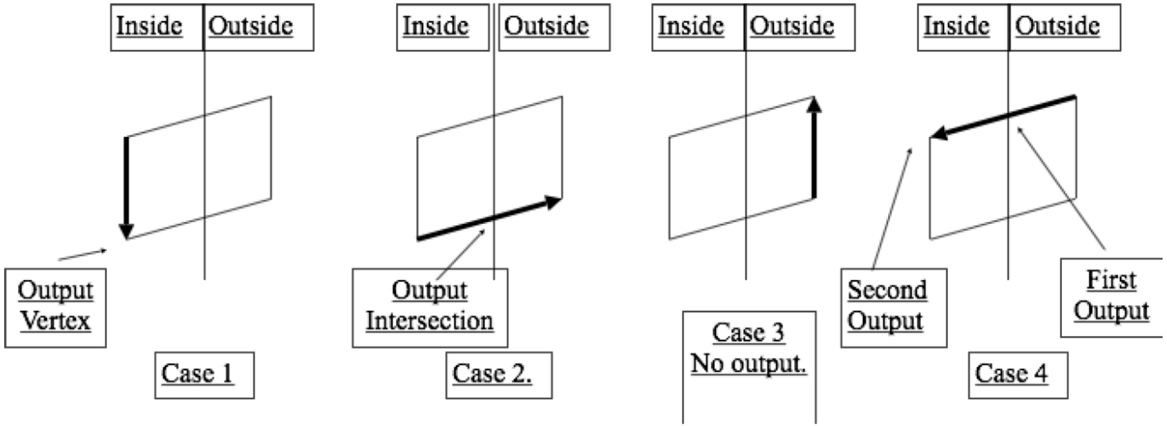
Traverse edges and divide into four types:



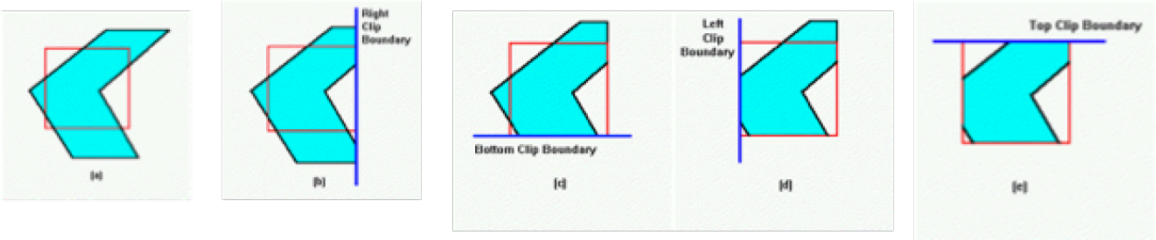
Sutherland-Hodgman algorithm

For each edge of the clipping rectangle:

- For each polygon edge between v_i and v_{i+1}



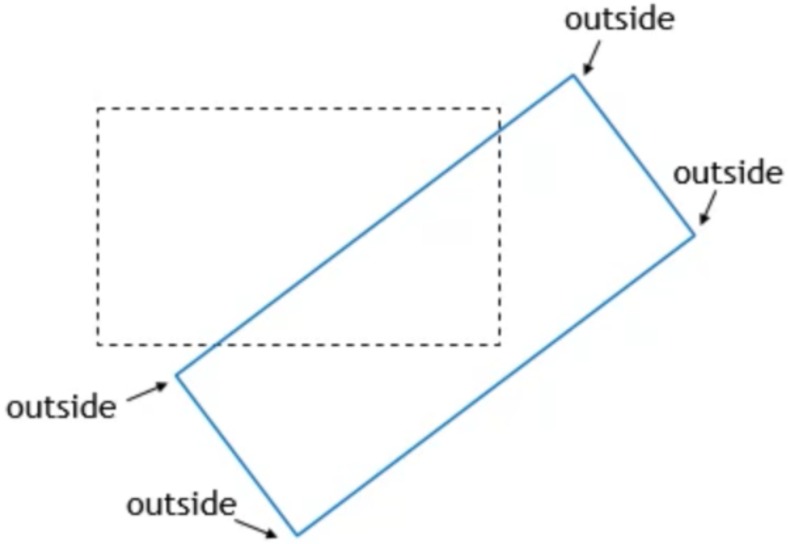
Example



(a)
Right
Clip
Boundary
(b)
Bottom Clip Boundary
(c)
Left
Clip
Boundary
(d)
Top Clip Boundary
(e)

Sutherland Hodgman Polygon Clipping

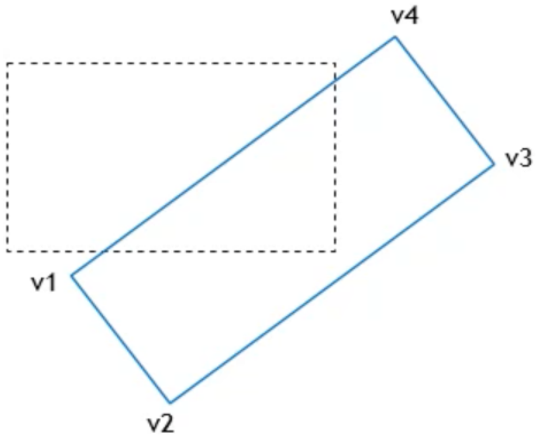
- What will happen here?



Sutherland Hodgman Polygon Clipping

- It is incorrect to consider a vertex as inside/outside of the clipping area

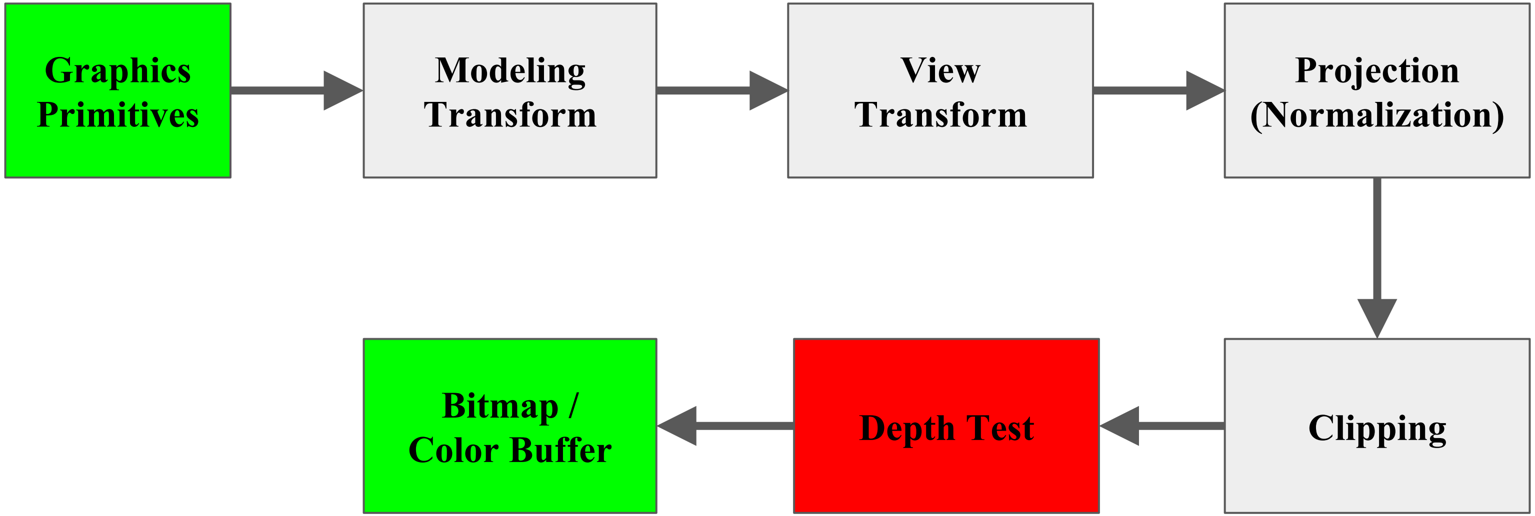
- Instead, for each vertex, tell whether it is **in the inner or outer side of each edge** of the clipping area



Sutherland Hodgman Polygon Clipping

- It is incorrect to consider a vertex as inside/outside of the clipping area
- Instead, for each vertex, tell whether it is in the inner or outer side of each edge of the clipping area
- **It also works in 3D world, where the clipping area consists of 6 surfaces, instead of 4 edges.**

Rendering Pipeline



HLHSR

- Target: generate the set of pixels that form the final image.

Algorithms (we’ll cover 2 this time)

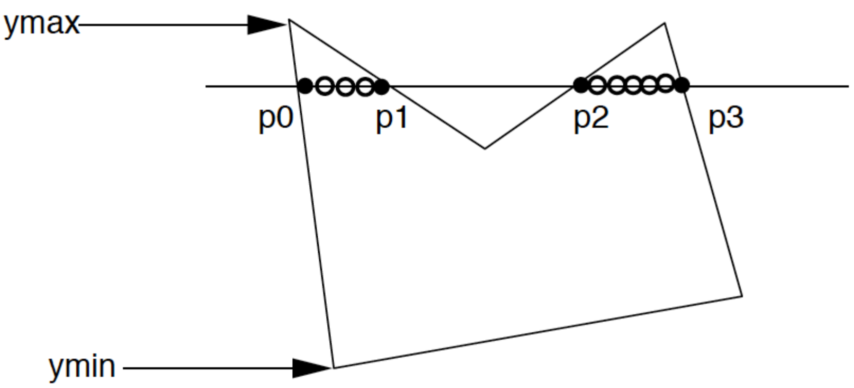
- **Scan-line Algorithm**
- **z-buffer Algorithm *(neat, simple and fast!)***
- Depth-Sort Algorithm
- Binary Space Partition (BSP) Trees
- Area-subdivision Algorithm

Scan-line Algorithm

- Intersect scanline with polygon edges
- Fill between pairs of intersections
- Basic algorithm:

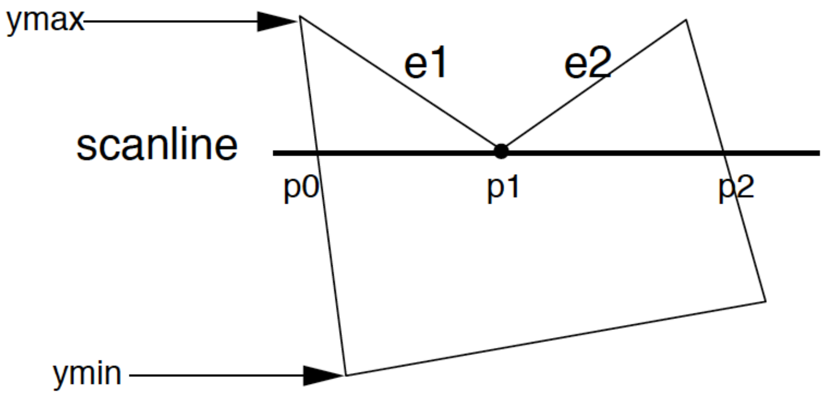
For y = ymin to ymax

1) intersect scanline y with each edge
2) sort interesections by increasing x
   [p0,p1,p2,p3]
3) fill pairwise (p0 –> p1, p2–> p3, ...)



Scan-line Algorithm: Special handling

● Intersection is an edge end point



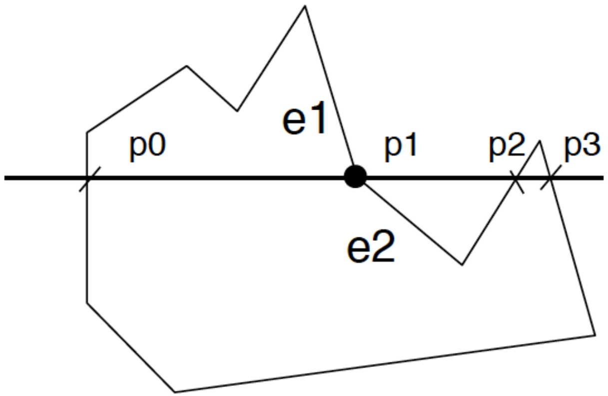
● Intersection points: (p0, p1, p2) ???

−> (p0,p1,p1,p2) so we can still fill pairwise

−> In fact, if we compute the intersection of the scanline with edge e1 and e2 separately, we will get the intersection point p1 twice. Keep both of the p1.

Scan-line Algorithm: Special handling (cont.)

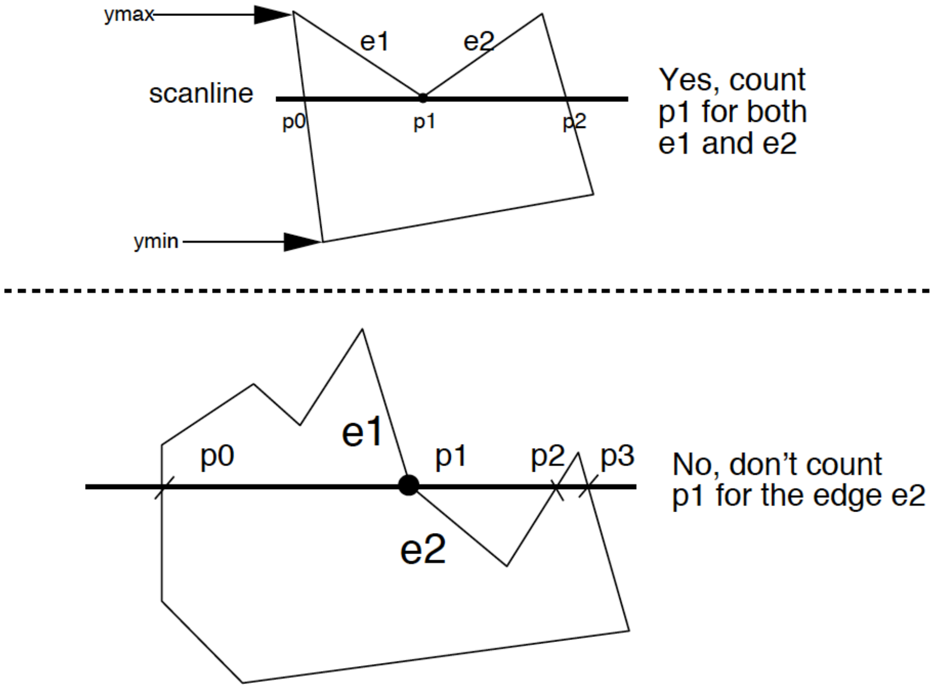
- Intersection is an edge end point



- However, in this case we don't want to count p1 twice (p0,p1,p1,p2,p3), otherwise we will fill pixels between p1 and p2, which is wrong

Scan-line Algorithm: Special handling (cont.)

- Rule: If the intersection is the ymin of the edge's endpoint, count it. Otherwise, don't.



z-buffer (a.k.a. depth-buffer) algorithm

- Initialization (2 buffers)

  z buffer (set to a value > 1) **z**

  color buffer (set to BG color) **c**

- For each polygon

  Scan convert

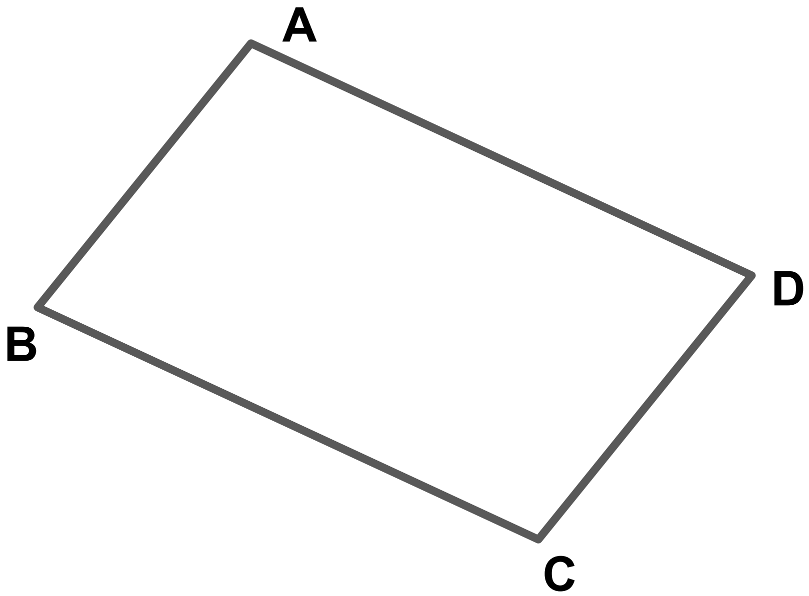
  For each pixel in the polygon

  if(z_poly(x,y)<**z**(x,y))

  z(x,y) = z_poly(x,y)

  c(x,y) = polygon_color

Obtain the Plain Equation from Polygon Vertices



z-buffer combined with scanline

- Calculating z_poly:
- Plane equation: $0 = A x + B y + C z + D$

  Solve for z: $z = (-A x – B y – D) / C$

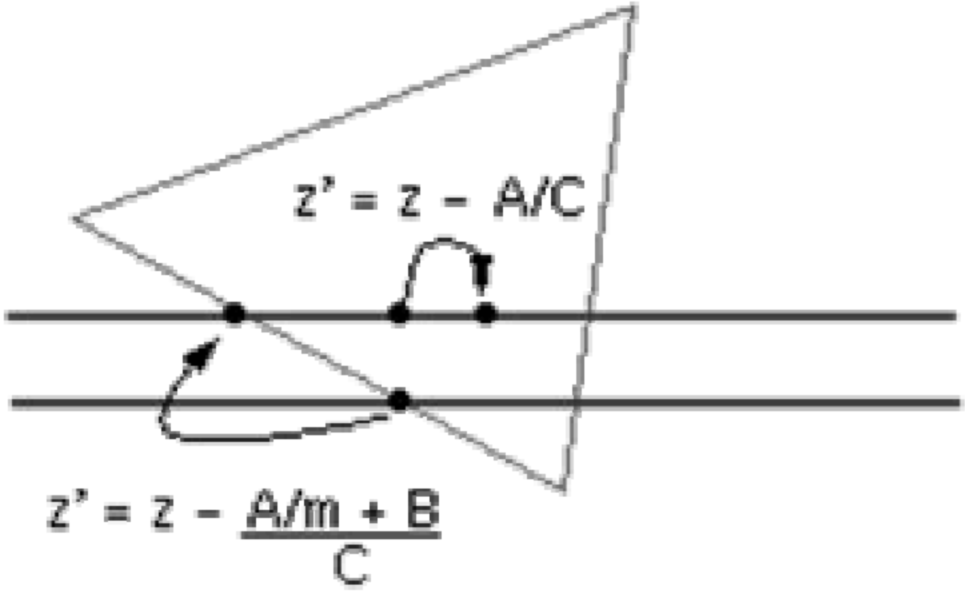
- Moving along a scanline, so want z at next value of x

  $Z' = (-A (x+1) – b y – D) / C$

  $Z' = z – A/C$

z-buffer combined with scanline

- For moving between scanlines , know x' = x + 1 / m
- The new left edge of the polygon is (x+1/m, y+1), giving z' = z - (A/m + B)/C



Q & A